Extracting Mathematical Semantics from LaTeX Documents

Jürgen Stuber[1] and Mark van den Brand[2,3]

[1] LORIA École des Mines de Nancy, 615 Rue du Jardin Botanique,
54600 Villers-lès-Nancy, France.
stuber@loria.fr

[2] Centrum voor Wiskunde en Informatica, Department of Software Engineering,
Kruislaan 413, NL-1098 SJ Amsterdam, The Netherlands,
Mark.van.den.Brand@cwi.nl

[3] Hogeschool van Amsterdam, Instituut voor Informatica en Electrotechniek,
Weesperzijde 190, NL-1097 DZ Amsterdam, The Netherlands

Abstract. We report on a project to use SGLR parsing and term rewriting with ELAN4 to extract the semantics of mathematical formulas from a LaTeX document and representing them in MathML. The LaTeX document we used is part of the Digital Library of Mathematical Functions (DLMF) project of the US National Institute of Standards and Technology (NIST) and obeys project-specific conventions, which contains macros for mathematical constructions, among them 200 predefined macros for special functions, the subject matter of the project. The SGLR parser can parse general context-free languages, which suffices to extract the structure of mathematical formulas from calculus that are written in the usual mathematical style, with most parentheses and multiplication signs omitted. The parse tree is then rewritten into a more concise and uniform internal syntax that is used as the base for extracting MathML or other semantical information.

1 Introduction

Mathematics is potentially an interesting field of application for the Semantic Web, as the underlying semantics is relatively clear and the main problem is to communicate it in a standard way, so it becomes machine usable, for example by computer algebra systems or theorem provers. However, today the semantics exists solely in the mind of the mathematician, who uses mathematical notation, typically in LaTeX, to communicate it to other mathematicians. The mathematical notation is originally two-dimensional in its graphical representation, take for example the use of subscripts and superscripts, or the notations for fractions and matrices. TeX reduces mathematical notation to a linear form, however as a natural language of humans it leaves out a lot of information that can be easily reconstructed by the human reader, for example the structure of expressions or their types. To enable machines to work with the semantics of mathematical formulas, there needs to be a notation that explicitly denotes expression structure and makes clear exactly which operations and objects are meant in a formula.

MathML [6] is an emerging standard for representing mathematical formulae, which however is much too verbose to be directly used by humans. For example, a short half-a-line formula in TEX corresponds to about half a page of MathML (see Figure 1). MathML comes in two varieties, Representation MathML which is targeted towards graphical representation for displaying or printing, and Content MathML designed to represent the mathematical semantics for computation or proving. Content MathML contains only basic high-school mathematics, for a wider variety of mathematical objects there is the OpenMath effort [16]. Both MathML and OpenMath address mathematical formulas in isolation, whereas OMDoc [10] allows to express the structure of mathematical documents, for example the relation between definitions, theorems and proofs.

We use ELAN4, which combines the rewriting of ELAN [3] with the powerful parser and development environment of ASF+SDF [7], to extract the semantics of mathematical formulas from a LATEX document and to generate a representation in MathML. In a first stage we use the SGLR parser [4] to parse the expression structure, which is then rewritten to an internal abstract representation, and finally to some form of MathML, currently Representation MathML.

Our project shows that it is feasible to extract mathematical formulas from a mathematical text written in a project-specific form of LATEX. In the particular project we worked on, the Digital Library of Mathematical Functions (DLMF) project (`http://dlmf.nist.gov/`) of the US National Institute of Standards and Technology (NIST), the subject matter was special functions, which has the properties that there are only few types (real and complex numbers and functions over these), and that there is a large body of macros for specific special functions. The task would be more difficult in subjects like algebra or logic, where there are more levels of abstraction and thus more ambiguity, and fewer easily identifiable mathematical notions.

Due to the time frame of only three months and lack of suitable tools we were not able to really investigate Content MathML, and we concentrated on Representation MathML instead. However, we want to emphasize that in contrast to other LATEX-to-MathML conversion tools [9,17], which transform a sequence of symbols in LATEX into a corresponding `mrow` element in MathML indiscriminately, we deduce the complete expression structure of the formula, and that it would thus would be much easier for us to derive Content MathML. To do this the main thing that is missing is the disambiguation between multiplication and function application, for example by type inference. We applied our tool separately to the sections of the sample chapter on Airy functions.[1] As this is only a proof of concept there are still parts missing, but, for example, we can treat the section on Scorer functions completely.[2] In particular, the current prototype cannot parse equations between expressions of function type,[3] the MathML representation for a large number of macros for special functions is still missing,

[1] `http://dlmf.nist.gov/Contents/AI/index.html`
[2] `http://dlmf.nist.gov/Contents/AI/AI.12.html`
[3] `http://dlmf.nist.gov/Contents/AI/AI.8_ii.html`

TEX:

```
\cos(\tfrac{1}{3}t^3+xt)
```

Internal abstract syntax:

```
apply(function("cos"),
  apply("(_)",
    apply("_+_",
      apply("__",
        frac("t",Int("1"),Int("3")),
        superscript(id(Simple,"t"),Int("3"))),
      apply("__",id(Simple,"x"),id(Simple,"t")))))
```

Representation MathML:

```
<mrow>
  <mo>cos</mo>
  <mo>&ApplyFunction;</mo>
  <mrow>
    <mo stretchy="false">(</mo>
    <mrow>
      <mrow>
        <mfrac displaystyle="false" scriptlevel="1">
          <mn>1</mn>
          <mn>3</mn>
        </mfrac>
        <mo>&InvisibleTimes;</mo>
        <msup><mi>t</mi><mn>3</mn></msup>
      </mrow>
      <mo>+</mo>
      <mrow><mi>x</mi><mo>&InvisibleTimes;</mo><mi>t</mi></mrow>
    </mrow>
    <mo stretchy="false">)</mo>
  </mrow>
</mrow>
```

Fig. 1. Blowup in the transformation from TEX to Representation MathML

and we currently do not use type inference for a more general disambiguation between multiplication and function application.

Since we currently do not have permission to publish parts of the DLMF chapter we worked on, we will only show small subformulas and point to the version published on the WWW [15] where appropriate. We also use examples from the predecessor of DLMF, the Handbook of Mathematical Functions [1], in particular Section 10.4 on Airy Functions.

2 Mathematical notation

Mathematical notation is a language invented by human mathematicians for communicating with other human mathematicians. As such it is a natural language, with a tendency to suppress information that can easily be deduced by the mathematician. For example, in contrast to programming languages which are designed to be parsed by machines, mathematical notation leaves out many parentheses and multiplication signs, and there is no global order of priorities to chose the right reading.

We looked at several mathematical texts to deduce rules for parsing mathematical formulas, first and foremost of course the chapter on Airy functions we were working on, the Handbook of Mathematical Functions [1], but also other books chosen for their variety and availability [2,5,8,12,18] to get a wider understanding of the problem. Wolfram [20] describes his understanding of mathematical notation, however we feel that it is not as standardized as he claims.

The omission of multiplication signs leads to an ambiguity between function application and multiplication, which can only be resolved using knowledge about the types. For example, $w(a+b)$ could mean that the function w is applied to $a+b$, or that w is multiplied by $a+b$.

The omission of parentheses complicates the parsing of expressions. In particular, for elementary transcendental functions, such as sin or log, parentheses around arguments are often omitted. Expressions such as $x \sin ax \cos bx$ are to be understood as $x(\sin(ax))(\cos(bx))$ where following factors are also part of the argument to the function, up to the next elementary transcendental functions. For example, this notation is used throughout the chapter on elementary transcendental functions in [1]. However, a formula like $\sin(p\pi)z^{-n/4}$ might also mean $(\sin(p\pi))z^{-n/4}$, i.e. in this case sin could be understood to have parentheses around its argument.[4] We resolve this by parsing a parenthesis immediately following an elementary transcendental function as its argument, excluding following factors. For the DLMF project, and in particular the chapter on Airy Functions, this seems to lead to correct parses. However, there are examples in other books [5, page 1069 (305)] where this will parse formulas incorrectly.

Similar conventions apply to big operators. For example, a sum operator extends typically to the next additive operation ($+$, $-$, $\pm$, $\mp$), including nested sums. Often this is made clear by the scope of the index variables of the sum,

[4] `http://dlmf.nist.gov/Contents/AI/about_AI.13.9.html`

for example the i in

$$\sum_i i \sum_j a_{ij}$$

shows that the scope of the first sum extends over the second. In any case, by the distributivity laws the equality

$$\sum_i (a_i \sum_j b_j) = (\sum_i a_i)(\sum_j b_j)$$

holds, so this ambiguity is usually not a problem. We do not currently treat other big operators, as their interaction with $\sum$ and other operations is not clear to us, and varies in the mathematical literature we surveyed.

For division it is rather unclear whether in $\sin a/b$ the b is part of the argument of sin, in practice this seems to depend on the particular a and b. In the DLMF this is resolved by always using macros for division, which makes this clear.

All of this can be expressed in an SDF grammar with the help of a hierarchy of sorts for various expressions. We will show such a grammar below for a fragment that contains the elementary transcendental functions.

3 Overall Structure

The technique that we use is to proceed in several stages, using the SGLR parser of ELAN4 to parse expressions and then rewriting them into the desired MathML representation (see Figure 2). Processing of a document begins by parsing it with the SGLR parser, which needs a relatively complex grammar that we describe in Section 4. The rest of the processing is done by several passes of rewriting. We first use a large rewriting system that parallels the grammar to rewrite the parse tree to an internal abstract syntax (Section 5), which is then made more uniform by successive rewriting phases (Section 6). From the final internal representation we produce an abstract version of MathML (Section 7), which is then rewritten to a parse tree for true XML by a small rewrite system (Section 8). From this the resulting XML can be created by the `unparse` tool that is part of ELAN4.

4 Parsing

We use the SGLR parser [4] in ELAN4, which permits to write unrestricted context-free grammars, even ambiguous ones, and has in addition a preference mechanism to choose certain parse trees if there are ambiguities. Grammars for SGLR are written in SDF, the Syntax Definition Formalism. We use preferences in the technique of "island parsing" [13], where a simple and loose subgrammar allows to parse the complete document in a rather flat and meaningless way, the "sea", and wherever possible more detailed subgrammars parse the parts that we are interested in, the "islands". In the case of this work the sea consists of the preamble and textual part, while the islands are the mathematical formulas. The SDF for the standard, non-mathematical part of the grammar is shown in Figure 3. The nonterminal `TeX-Element` is extended in `Math-Env` by environments

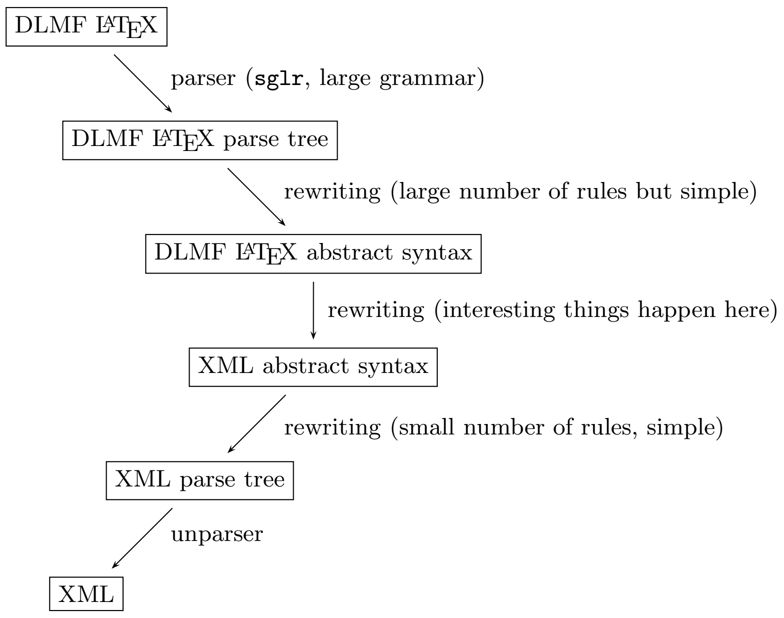
DLMF LaTeX
parser (sglr, large grammar)
DLMF LaTeX parse tree
rewriting (large number of rules but simple)
DLMF LaTeX abstract syntax
rewriting (interesting things happen here)
XML abstract syntax
rewriting (small number of rules, simple)
XML parse tree
unparser
XML

Fig. 2. Overview of the system

for mathematical formulas, where all the mathematical parsing takes place. The mathematical part consists in turn of several environments which are specific to the DLMF project, which contain equations or mathematical formulas together with for example labels and comments.

The central component of the grammar for mathematical formulas is the grammar module Equation, which describes the rules according to which mathematical expressions can be formed. This grammar is rather large, it has about 100 productions, so we cannot present it here. Instead we present a fragment of its core that describes basic arithmetic and elementary transcendental functions in Figure 4. As said this is only a small fraction, the real grammar contains many more rules, for example for fractions, differentials, integrals and other, less well-known mathematical operators. Our example grammar relies on the presence of grammars for `AddOp`, `MultOp`, `Function`, `Number` and `Variable`, which are of a very simple structure, either a few lexical definitions or dictionaries of functions. A small example dictionary is shown in Figure 5. In the real grammar the dictionaries are much larger, for example there are about 200 macros for specific functions in the DLMF latex package.

5 Rewriting to Abstract Syntax

The result of parsing is a parse tree that conserves all the syntactical details of a mathematical formula. In particular, each grammar rule becomes a function symbol in the parse term, even though several grammar rules may represent

```
module Latex-Document

imports
    Math-Env
    Layout

exports
  sorts
    LaTeX-Document
    Doc-Class
    Tex-Element
    Tex-Token
    Comment
    Text-Token
    Macro
    Special-Macro
    Bracket-Struct

  lexical syntax
    [A-Za-z0-9\-]+              -> Doc-Class

    [\\] [a-zA-Z]+ [\\]?        -> Macro
    [\\] ~[a-zA-Z]              -> Special-Macro
    ~[\\\%\ \n\{\}\[\]\#\$]+    -> Text-Token
    [\#][0-9]+                  -> Text-Token
    "%" ~[\n]* [\n]             -> Comment

  context-free restrictions
    Macro -/- [A-Za-z]
    Text-Token -/- ~[\\\%\ \n\{\}\[\]\#\$]

  context-free syntax
    "\\documentclass{" Doc-Class "}" TeX-Element*
    "\\begin{document}" TeX-Element* "\\end{document}" -> LaTeX-Document

    Comment                     -> TeX-Token
    Text-Token                  -> TeX-Token
    Macro                       -> TeX-Token
    Bracket-Struct              -> TeX-Token
    Special-Macro               -> TeX-Token

    TeX-Token                   -> TeX-Element

  context-free syntax
    "{" TeX-Element* "}"        -> Bracket-Struct
    "[" TeX-Token+ "]"          -> Bracket-Struct
```

Fig. 3. Top part of island grammar

```
module Expression

imports
    Layout
    Dictionaries

exports
    sorts
        Expression SumProduct ETFProduct SumApplication ETFApplication
        SimpleProduct Power Base SumOp ETFunction

    context-free syntax
                         ETFProduct -> Expression
                   AddOp ETFProduct -> Expression
        Expression AddOp ETFProduct -> Expression

        SimpleProduct                    -> ETFProduct
                          ETFApplication -> ETFProduct
        ETFProduct MultOp ETFApplication -> ETFProduct

        ETFunction SimpleProduct      -> ETFApplication
        ETFunction ETFApplication     -> ETFApplication

                             Power -> SimpleProduct
        SimpleProduct MultOp Power -> SimpleProduct

        Base                         -> Power
        Base "^" "{" Expression "}" -> Power

        Number                       -> Base
        Variable                     -> Base
        "(" Expression ")"          -> Base
        Function "(" Expression ")" -> Base

        "\\sin"                        -> ETFunction
        "\\log" "_" "{" Expression "}" -> ETFunction

        ETFunction -> Function {prefer}
```

Fig. 4. Simplified grammar for mathematical expressions

```
module Dictionaries

sorts
    Integer Number Variable Function LogFunction AddOp MultOp SumOp

exports
    lexical syntax
        [\-]?[1-9][0-9]* -> Integer

    context-free syntax
        Integer -> Number

        "x" -> Variable
        "y" -> Variable
        "i" -> Variable
        "n" -> Variable

        "f" -> Function
        "g" -> Function

        "+" -> AddOp
        "-" -> AddOp

            -> MultOp
        "*" -> MultOp
```

Fig. 5. Dictionary for the example grammar

```
[] #to_term_expression(#$Expression_1# #$PlusOp# #$Expression_2#)
   => apply(#$Term_op#,#$Term_1#,#$Term_2#)
   where #$Term_op# := #to_term_plus_op(#$PlusOp#)
   where #$Term_1# := #to_term_expression(#$Expression_1#)
   where #$Term_2# := #to_term_expression(#$Expression_2#)
```

Fig. 6. Typical rule to rewrite to abstract syntax

the same mathematical object. For example, in the full grammar the hierarchy of sorts leads to 7 rules for multiplication. This would make it extremely hard to work with, as each of the redundant cases will need to be treated separately. We chose to obtain a more uniform representation as the first step, rewriting the parse tree to an abstract internal representation that is more uniform and closely follows the mathematical structure. The abstract syntax consists of variable-arity terms in prefix notation, with optional annotations. Atoms are either constants or strings. For example, `plus`, `apply(plus,"x","1")` and `mo("("){[xml_attribute(stretchy),"false"]}` are terms in the abstract syntax. The optional annotations consist of a list of pairs of terms in braces; we use it mostly to represent XML attributes. The abstract syntax was chosen so that it is a subset of the textual representation of ATerms [19].

The rewriting system that converts parse tree to abstract syntax parallels the grammar, as grammar rules become function symbols in the parse trees, so we essentially need a rule for each of these function symbols. A typical rule is shown in Figure 6. `#to_term_expression` is the function that converts parse trees of sort (resp. nonterminal) `Expression` to abstract syntax. The general form of the ELAN4 rules that we use (there are other features such as strategies that we do not use currently) is

$$\texttt{[]}\ l \texttt{ => } r \quad \texttt{where } t_1 \texttt{ := } s_1 \ \ldots \ \texttt{where } t_n \texttt{ := } s_n$$

where l, r, s_i and t_i are terms. If l matches a subterm of the current term then the variables in l are bound, and the terms in the right-hand sides s_i of the where clauses are successively rewritten to normal form and then matched against the corresponding left-hand side t_i. If this match fails the rule is not applied, otherwise the variables in t_i are bound and the process continues. At the end l is replaced by r in the current term, with the variables in r instantiated by their bound values. In the example variables have the syntax `#$`*Sort_Suffix*`#` where the optional suffix distinguishes several variables of the same sort. The `#` helps to distinguish them from abstract terms and TEX-text.

This example illustrates that in the internal syntax we represent most mathematical expressions in the form `apply(`*operation*`,`*arguments*`)`, except for fractions, large operators, differentials and integrals, which have their special representation.

6 Improving the Abstract Syntax

The abstract syntax representation obtained in the previous step is still very close to the original grammar, and needs to be refined to exhibit all the information

```
rules

  [] #improve_abstract_syntax(#$Term#)
     => #$Term8#
     where #$Term0# := #collate_tex_text(#$Term#)
     where #$Term1# := #move_macro_argument(#$Term0#)
     where #$Term2# := #transform_text_envs(#$Term1#)
     where #$Term3# := #transform_references(#$Term2#)
     where #$Term4# := #transform_headings(#$Term3#)
     where #$Term5# := #transform_user_macros(#$Term4#)
     where #$Term6# := #transform_equation_envs(#$Term5#)
     where #$Term7# := #transform_text(#$Term6#)
     where #$Term8# := #transform_preamble(#$Term7#)
```

Fig. 7. Phases for rewriting the abstract syntax

needed in subsequent steps in a convenient format. We use several passes that traverse the complete term and each does a particular operation on the tree (see Figure 7). First, we combine adjacent texts into one to substantially reduce the term size, and we attach arguments to macros, which in the grammar are braces that follow macros. Currently we do not do more sophisticated semantical processing, such as type inference, however this could easily be extended.

The remaining phases `#transform_X` are concerned with generating abstract XML (XHTML and Representation MathML) for output, which we discuss in the following section.

7 Rewriting to Abstract Representation MathML

As the final step within abstract syntax we generate an abstract version of XML for output. XML elements are represented as function applications, text nodes as strings and attributes as annotations. For example, the abstract syntax term

```
mo("("){[xml_attribute(stretchy),"false"]}
```

represents the XML

```
<mo stretchy="false">(</mo>.
```

We also have a special notation `XML_Reference(`*String*`)` to represent character references, for example `XML_Reference("int")` becomes `&int;`.

With this representation of XML in place it is straightforward to write rules that transform our internal representation. Figure 8 contains the fragment of the code that handles the various cases of `apply`, together with a few lines from the dictionary rules to illustrate their format. Here we use the `where` clauses of the ELAN4 rules to distinguish the different operation types, in order to generate different output. For example, in

```
where infix(#$Term_mo#) := #to_mrep_op(#$Term_op#)
```

```
[] #to_mrep(apply(#$Term_op#, #$Term_1#, #$Term_2#))
   => mrow(#to_mrep(#$Term_1#),#$Term_mo#,#to_mrep(#$Term_2#))
   where infix(#$Term_mo#) := #to_mrep_op(#$Term_op#)

[] #to_mrep(apply(#$Term_op#, #$Term#))
   => mrow(#$Term_mo#,#to_mrep(#$Term#))
   where prefix(#$Term_mo#) := #to_mrep_op(#$Term_op#)

[] #to_mrep(apply(#$Term_op#, #$Term#))
   => mrow(#to_mrep(#$Term#),#$Term_mo#)
   where postfix(#$Term_mo#) := #to_mrep_op(#$Term_op#)

[] #to_mrep(apply(#$Term_op#, #$Term#))
   => mrow(#$Term_mol#,#to_mrep(#$Term#), #$Term_mor#)
   where fence(#$Term_mol#,#$Term_mor#) := #to_mrep_op(#$Term_op#)

[] #to_mrep(apply(#$Term_op#, #$Term#))
   => mrow(#$Term_mo#,mo(XML_Reference("ApplyFunction")),#to_mrep(#$Term#))
   where et_function(#$Term_mo#) := #to_mrep_op(#$Term_op#)

[] #to_mrep(apply(#$Term_op#, #$Term#))
   => mrow(#$Term_mo#,
           mo(XML_Reference("ApplyFunction")),
           mo("("),
           #to_mrep(#$Term#),mo(")"))
   where function(#$Term_mo#) := #to_mrep_op(#$Term_op#)

[] #to_mrep(apply("\sqrt", #$Term#))
   => msqrt(#to_mrep(#$Term#))

[] #to_mrep(apply("\sqrt", #$Term_exp#, #$Term#))
   => msqrt(#to_mrep(#$Term#),#to_mrep(#$Term_exp#))

[] #to_mrep_op("_+_") => infix(mo("+"))
...
[] #to_mrep_op("+_") => prefix(mo("+"))
...
[] #to_mrep_op("_!") => postfix(mo("!"))
...
[] #to_mrep_op(function("sin")) => et_function(mo("sin"))
...
[] #to_mrep_op(function("AiryAi")) => function(mo("Ai"))
...
[] #to_mrep_op("(_)") => fence(mo("("){[xml_attribute(stretchy),"false"]},
                               mo(")"){[xml_attribute(stretchy),"false"]})
[] #to_mrep_op("\left(_\right)") => fence(mo("("),mo(")"))
[] #to_mrep_op("|_|") => fence(mo("|"),mo("|"))
```

Fig. 8. Transformation to MathML of apply

```
[] <#term_to_XML_element(#$Identifier#(#$Term,+#) #$Annotation?#)#>
  => <$QName $Attributes><#terms_to_XML_nodes(#$Term,+#)#></$QName>
  where $QName := #term_to_qname(#$Identifier#)
  where $Attributes := <#opt_annotation_to_attributes(#$Annotation?#)#>
```

Fig. 9. A core rule in the transformation from abstract syntax to XML

the rewriting of the term on the right-hand side results in a normal form. If this normal form has the function symbol `infix` at the root it matches the left-hand side, its argument gets bound to the variable `#$Term_mo#` and the rule is executed. In this way it is easy to write dictionaries for a large number of operators.

8 Generating XML

We have written an SDF grammar for XML with namespaces that can be used both for parsing and generating XML documents, which has the side effect that generated documents must be syntactically correct. We use a small rewrite system of 27 rules to generate XML from internal abstract syntax, which is not specific to MathML. As an example we show the core rule that creates an XML element from a function symbol in Figure 9. There are more rules for traversing the term and for the other XML nodes. In particular the rules for traversing make heavy use of list matching, since for example nodes below an element are described in the grammar by a `*` operator. The strange syntax with `<`, `#` and `$` characters is again used to ensure that operations and variables are not parsed as TEX or XML text.

Using `unparse` on the resulting XML parse tree produces an XML document that can be passed to other tools as input, in our case for example Mozilla for Representation MathML display.

9 Performance

Of the 17 sections in the DLMF sample chapter on Airy functions we can handle the mathematical formulas completely in 6, partially in 5 (without the transformation to MathML), and 6 remain incomplete.

The grammar currently contains approximately 1000 productions, of which ca. 350 are dictionaries. There are about 550 rewrite rules. There are fewer rewrite rules than grammar rules, partly because dictionaries can be treated uniformly by manipulating literals, and partly because it is still incomplete with respect to the grammar.

On a 1.8GHz Pentium 4 compiling the grammar and rules takes about a minute, while parsing is relatively fast, on the order of a few seconds for the complete chapter. The result is a parse tree of several hundred thousand nodes. Rewriting it is comparatively slow, on the order of several minutes, since it is done by an interpreter. We do not currently have a compiler for ELAN4.

We feel that the limit of what can be achieved with ELAN4 is not yet reached.

10 Conclusion

SGLR permits a very flexible syntax, which allows to represent both LaTeX and XML directly. However, having these two markup languages, where almost every input string except for some escape characters is legal, lead to problems in correctly parsing the rewrite rules. These were overcome by designing a special syntax for internal variables and function symbols, as these can also be chosen freely.

Our project shows that parsing mathematics in the form of LaTeX documents written to project-specific rules is feasible, but due to the variations in notation the grammar needs to be engineered specifically for the project, or even for different chapters written by different mathematicians (e.g. the chapter on elementary transcendental functions and on Airy functions).

In this kind of work there will also always be some part that cannot treated automatically. For example, the example chapter contains the formula

$$\int\int\cdots\int f(t)\,(dt)^n,$$

which, even if we could parse it, we would not know how to represent in Content MathML.

The use of ELAN4 is not a prerequisite but was a convenient vehicle due to its combination with SDF. We could also have used ASF, since we currently do not use ELAN's strategies, however these might be useful for type inference. Writing an equivalent grammar with, for instance, LEX+YACC will probably next to impossible. It would also have been possible to keep the SGLR parser and the grammar, but to use other tools for the transformation, in particular JAVA tools like TOM [14] or JJForester [11]. The advantage of using ELAN4 or ASF over these is that use of user-defined syntax for both the input format as well as the output format ensures syntactically correct results. Another possible route would be to translate the parse trees into XML and to express the transformation in XSLT.

Parsing mathematical formulas in LaTeX documents is a real challenge. In this paper we only address the translation to Representation MathML, due to time constraints. The translation to Content MathML is a next step in this project and would create a link with computer algebra systems like Mathematica or Maple. We feel that to generalize and extend these results further some of the implicit mathematical semantic information, in particular type information, needs to be encoded in the document and used by more semantically directed parsing techniques.

11 Acknowledgments

We thank Gaurav Kwatra from IIT Delhi, who has worked with us on this project as a summer intern, Claude Kirchner and Michael Kohlhase for initiating the project, and Bruce R. Miller from NIST for making available the sample chapter.

References

1. Milton Abramowitz and Irene Stegun, editors. *Handbook of Mathematical Functions.* National Bureau of Standards, USA, 1964.
2. Martin Aigner and Günter M. Ziegler. *Proofs from THE BOOK.* Springer, 1998.
3. Peter Borovanský, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. ELAN *V 3.3 User Manual.* LORIA, Nancy (France), third edition, December 1998.
4. M.G.J. van den Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In *Compiler Construction (CC'02)*, LNCS 2304, pages 143–158, Grenoble, France, 2002. Springer. See `http://www.cwi.nl/projects/MetaEnv/`.
5. I. N. Bronstein, K. A. Semendjajew, G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik.* Harri Deutsch, 5th edition, 2000.
6. David Carlisle, Patrick Ion, Robert Miner, and Nico Poppelier, editors. *Mathematical Markup Language (MathML) Version 2.0.* W3C, 21 February 2001. `http://www.w3.org/TR/2001/REC-MathML2-20010221/`.
7. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific, 1996.
8. Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics.* Addison-Wesley, 2nd edition, 1994.
9. Eitan M. Gurari. Tex4ht: Latex and tex for hypertext. `http://www.cis.ohio-state.edu/~gurari/TeX4ht/mn.html`.
10. Michael Kohlhase. OMDoc: An open markup format for mathematical documents, 2003. See `http://www.mathweb.org/omdoc/`.
11. T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. *Science of Computer Programming*, 47(1):59–87, November 2002.
12. Serge Lang. *Algebra.* Addison-Wesley, Reading, Mass., 3rd edition, 1993.
13. Leon Moonen. Generating robust parsers using island grammars. In *Proc. 8th Working Conf. on Reverse Engineering*, pages 13–22. IEEE Computer Society Press, 2001.
14. Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, May 2003.
15. Frank W. J. Olver. *Digital Library of Mathematical Functions*, chapter Airy and Related Functions. National Institute of Standards and Technology, 2001. `http://dlmf.nist.gov/Contents/AI/index.html`.
16. OpenMath. `http://www.openmath.org/`.
17. John Plaice and Yannis Haralambous. Produire du MathML et autres ...ML à partir d'Ω : Ω se généralise. In *Cahiers GUTenberg no 33 — actes du congrès GUT'99*, Lyon, May 1999.
18. Günter Scheja and Uwe Storch. *Lehrbuch der Algebra.* B. G. Teubner, Stuttgart, 2nd edition, 1994.
19. M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.
20. Stephen Wolfram. Mathematical notation: Past and future. Transcript of a keynote address presented at MathML and Math on the Web: MathML International Conference 2000. Available at `http://www.stephenwolfram.com/publications/talks/mathml/`.